AMENDMENTS TO THE CLAIMS

Claims 1-3. (Canceled)

4. (Currently amended) A method of verifying a program fragment downloaded onto a reprogrammable ~~on-board~~ embedded system, ~~such as a microprocessor card~~ equipped with a rewritable memory, a microprocessor and a virtual machine equipped with an execution stack and with operand registers, said program fragment consisting of an object code and including at least one subprogram[,] consisting of a series of instructions manipulating said operand registers, ~~by the microprocessor of the on-board system by way of a virtual machine equipped with an execution stack and with operand registers manipulated by these instructions, and~~ said microprocessor and virtual machine making it possible to interpret [this] said object code, said ~~on-board~~ embedded system being interconnected to a reader, ~~characterized in that said method, following~~ wherein subsequent to the detection of a downloading command and the storage of said object code constituting [this] said program fragment in said rewritable memory, ~~consists,~~ said method, for each subprogram, includes:

~~a) in carrying out a stage of~~ initializing the type stack and the table of register types [by] through data representing the state of the virtual machine at the starting of the execution of [the] said temporarily stored object code;

~~b) in~~ carrying out a verification process of said temporarily stored object code instruction by instruction, by discerning the existence, for each current instruction, of a target, a branching—instruction target, a target of an exception—handler call or a target of a subroutine call, and, said current instruction being the target of a branching instruction, said verification process consisting in verifying that the stack is empty and rejecting the program fragment otherwise;

~~c) in~~ carrying out a verification process and an updating of the effect of said current instruction on the data types of said type stack and of said table of register types[,];

~~on the basis of the existence of a branching instruction target, of a target of a subroutine call or of a target of an exception handler call~~, said verification process being successful when the table of register types is not modified in the course of a verification of all the instructions, and [the] said verification process being carried out instruction by instruction until the table of register types is stable, with no modification being present, the verification process being interrupted and said program fragment being rejected, otherwise.

5. (Currently amended) The [verification]method [as claimed in]of claim 4, [characterized in that]wherein the variable types which are manipulated during [the]said verification process include at least:

[—]class identifiers corresponding to object classes which are defined in the program fragment;

[-]numeric variable types including at least a type ~~short,~~ for an integer coded on [p] a given number of bits, designated as short type, and a type ~~retaddr~~ for the return address of a jump instruction [JSR], designated as a return address type;

[- a type null relating to] references of null objects designated as null type;

[— a] object type ~~object~~ relating to objects designated as object type;

[-]a first specific type [⊥,] representing the intersection of all the types and corresponding to the zero value [0, nil], designated as the intersection type;

[-]a second specific type [T,] representing the union of all the types and corresponding to any type of value, designated as the union type.

6. (Currently amended) The m[M]ethod ~~as claimed in~~ of claim 5, ~~characterized in that~~ wherein all said variable types verify a subtyping relation:

[object ϵ T] object type belongs to the union type;

[short, retaddr ϵ T] short type and return address type belong to the union type;

[$\perp$ ϵ null, short, retaddr] the intersection type belongs to null type, short type or return address type.

Claim 7. (Canceled)

8. (Currently amended) The method [as claimed in one] of claim[s] 4 [to 7], [characterized in that when]wherein said current instruction [is] being the target of a subroutine call, said verification process [verifies]consists in:

verifying that the previous instruction to said current instruction is an unconditional branching, a subroutine return or a [raising]withdrawal of an exception[,]; and ~~said verification process, in the case of a positive verification, proceeding to~~ reupdat[e]ing the stack of variable types by an entity of [retaddr]the return address type, formed by the return address of the subroutine, in case of a positive verification process; and,

[the]rejecting said program fragment in case said verification process is failing, ~~and the program fragment being rejected~~ otherwise.

9. (Currently amended) The method [as claimed in one] of claim[s] 4 [to 8], [characterized in that when the]wherein said current instruction [is] being the target of an exception handler, said verification process [verifies] consists in:

verifying that the previous instruction to said current instruction is an unconditional branching, a subroutine return or a [raising] withdrawal of an exception[,]; [said verification process, in] and

reupdating the type stack, by entering the exception type, in [the] case of a positive verification process; ~~proceeding to reupdate the type stack by entering the exception type, and the verification process failing and the program fragment being rejected~~ and

rejecting said program fragment in case of said verification process is failing, otherwise.

10. (Currently amended) The method [as claimed in one] of claim[s] 4 [to 9], [characterized in that when the]wherein said current instruction [is]being the target of multiple incompatible branchings, [the]said verification process is fail[s]ed and [the]said program fragment is rejected.

11. (Currently amended) The method [as claimed in one] of claim[s] 4 [to 10], [characterized in that when the]wherein said current instruction [is]being not the target of any branching, [the] said verification process [continues]consists in continuing by passing to an update of the type stack.

12. (Currently amended) The method [as claimed in one] of claim[s] 4 [to 11], [characterized in that the stage]wherein said step of verification of the effect of the current instruction on the type stack includes, at least:

[- a stage of]verifying that the type execution stack includes at least as many entries as the current instruction includes operands;

[- a stage of]unstacking and [of] verifying that the types of the entries at the top of the stack are subtypes of the types of the operands types of the operands of [this]said current instruction;

[- a stage of]verifying the existence of a sufficient memory space on the types stack to proceed to stack the results of [the]said current instruction;

[- a stage of] stacking on the stack data types which are assigned to these results.

13. (Currently amended) The method [as claimed in]of claim 12, [characterized in that when the]wherein said current instruction [is]being an instruction to read a register of a given address [n], [the] said verification process consists in:

[— in]verifying the data type of the result of [this] a corresponding reading, by reading [the]an entry [n] at said given address in the table of register types;

[— in]determining the effect of [the]said current instruction on the type stack by unstacking the entries of the stack corresponding to the operands of [this]said current instruction and by stacking the data type of [this]said result.

14. (Currently amended) The method [as claimed in]of claim 12, [characterized in that when the] wherein said current instruction [is]being an instruction to write to a register of a given address [m], [this]said verification process consists in:

[— in]determining the effect of the current instruction on the type stack and the given type [t] of the operand which is written in this register [of]at said given address[m];

[— in]replacing the type entry of the table of register types at said given address [m]by the type immediately above the previously stored type and above the given type [t]of the operand which is written in this register [of] at said given address [m].

15. (Currently amended) A method of transforming an object code of a program fragment including a series of instructions, in which the operands of each instruction belong to the data types manipulated by [this]said instruction, the execution stack does not exhibit any overflow phenomenon, and for each branching instruction, the type of the stack variables at [this] a corresponding branching is the same as [at the]that of targets of this branching, into a standardized object code for this same program fragment,~~in which the operands of each instruction belong to the data types manipulated by this instruction, the execution stack does not exhibit any overflow phenomenon, the execution stack is empty at each branching instruction and at each branching target instruction, characterized in that this method consists,~~ wherein, for all the instructions of said object code, said method consists in:

[- in]annotating each current instruction with the data type of the stack before and after execution of [this]said current instruction, with the annotation data being calculated by means of an analysis of the data stream relating to [this]said current instruction;

[- in]detecting, within said instructions and within each current instruction, the existence of branchings, or respectively of branching-targets, for which said execution stack

is not empty, [the]said detecti[on]ng operation being carried out on the basis of the annotation data of the type of stack variables allocated to each current instruction[,]; and in [the presence]case of detection of a non-empty execution stack,

[- in]inserting instructions to transfer stack variables on either side of [these]said branchings or of [these]said branching targets[,] respectively, in order to empty the contents of the execution stack into temporary registers before [this]said branching and to reestablish the execution stack from said temporary registers after [this]said branching[,]; and [in]not inserting any transfer instruction otherwise, [making it possible]said method allowing thus to obtain a standardized object code for [this]said same program fragment, in which the operands of each instruction belong to the data types manipulated by said instruction, the execution stack does not exhibit any overflow phenomenon, the execution stack is empty at each branching instruction and at each branching—target instruction, in the absence of any modification to the execution of said program fragment.

16. (Currently amended) A method of transforming an object code of a program fragment including a series of instructions, in which the operands of each instruction belong to the data types manipulated by [this]said instruction, and an operand of given type written into a register by an instruction of this object code is reread from this same register by another instruction of [this]said object code with the same given data type, into a standardized object code for this same program fragment, ~~in which the operands of each instruction belong to the data types manipulated by this instruction, the same data type being allocated to the same register~~

~~throughout said standardized object code, characterized in that this method consists,~~wherein for all the instructions of said object code, said method consists in:

[- in]annotating each current instruction with the data type of the registers before and after execution of [this]said current instruction, with the annotation data being calculated by means of an analysis of the data stream relating to [this]said instruction;

[- in]carrying out a reallocation of [the]said registers, by detecting the original registers employed with different types, [by]dividing these original registers into separate standardized registers, with one standardized register for each data type used, and

reupdating the instructions which manipulate the operands which use said standardized registers;

said method allowing thus to obtain said standardized object code for this same program fragment in which the operands of each instruction belong to the data types manipulated by said instruction, the same data type being allocated to the same register throughout said standardized object code.

17. (Currently amended) The method [as claimed in] of claim 15, ~~characterized in that the stage consisting in~~wherein said detecting[,] within said instructions and within each current instruction[,] of the existence of branchings, or respectively of branching targets, for which the execution stack is not empty, [consists, following] after detection of each corresponding instruction of given rank [i]consists in:

[- in]associating with each instruction of said given rank [i]a set of new registers, one new register being associated with each stack variable which is active at this instruction; and

[- in]examining each detected instruction of said given rank [i]and [in] discerning the existence of a branching target or branching, respectively[,]; and, in the case where the instruction of said given rank [i] is a branching target and that the execution stack at this instruction is not empty,

[●]for every preceding instruction, of rank [i—1,]preceding said given rank and consisting of a branching, a [raising]withdrawal of an exception or a program return, [the]said detected instruction of said given rank [i] being accessible only by a branching,

[●●] [in]inserting a set of loading instructions [load]to load from the set of new registers before said detected instruction of said given rank[i], with a redirection of all branchings to the detected instruction of said given rank [i]to the first inserted loading instruction[load]; and

[●]for every preceding instruction, of rank [i—1] preceding said given rank, continuing in sequence, [the]said detected instruction of said given rank [i]being accessible simultaneously [by]from a branching and from [the]said preceding instruction of rank [i-l] preceding said given rank,

[●●] [in]inserting a set of backup instructions [store] to back up to the set of new registers before the detected instruction of said given rank[i], and a set of loading instructions [load]to load from this set of new registers, with a redirection of all

the branchings to the detected instruction of said given rank [i]to the first inserted loading instruction[load], and, in the case where said detected instruction of said given rank [i] is a branching to a given instruction,

[●]for every detected instruction of said given rank [i]consisting of an unconditional branching,

[●●][in]inserting, before the detected instruction of said given rank[i], multiple backup instructions [store], a backup instruction being associated with each new register; and

[●]for every detected instruction of said given rank [i]consisting of a conditional branching instruction, and for a given number [m > 0]greater than zero of operands manipulated by [this]said conditional branching instruction,

[●●] [in]inserting, before [this] said detected instruction of said given rank[i], a permutation instruction, [swap—x,]at the top of the execution stack of the [m]operands of the detected instruction of said given rank [i]and the [n]following values, [this]the corresponding permutation operation [making it possible]allowing thus to collect at the top of the execution stack [the n] said following values to be backed up in the set of new registers[,]; and

[●●] [in]inserting, before the instruction of said given rank[i], a set of backup instructions [store]to back up to the set of new registers[,]; and

[●●] [in]inserting, after the detected instruction of said given rank[i], a set of load instructions [load] to load from the set of new registers.

18. (Currently amended) The method [as claimed in]of claim 16, [characterized in that]wherein the [stage]step consisting in reallocating registers by detecting the original registers employed with different types consists in:

[— in]determining the lifetime intervals of each register;

[— in]determining the main data type of each lifetime interval, the main data type of a lifetime interval [j]for a given register [r]being defined by the upper bound of the data types stored in [this] said given register [r]by the backup instructions [store]belonging to [the]said lifetime interval[j];

[- in]establishing an interference graph between the lifetime intervals, [this]said interference graph consisting of a non-oriented graph of which each peak consists of a lifetime interval, and of which the arcs between two peaks [j_1 and j_2]exist if [a]one of the peaks contains a backup instruction addressed to the register of the other peak or vice versa;

[- in]translating the uniqueness of a data type which is allocated to each register in the interference graph, by adding arcs between. all pairs of peaks of the interference graph while two peaks of a pair of peaks do not have the same associated main data type;

[- in]carrying out an instantiation of the interference graph, by assigning to each lifetime interval a register number, in such a way that different register numbers are assigned to two adjacent life time intervals in [the]said interference graph.

Claim 19. (Canceled)

20. (Currently amended) An [on-board]embedded system which can be reprogrammed by downloading program fragments, said embedded system including a least one microprocessor, one random-access memory, one input/output module, one electrically reprogrammable nonvolatile memory and one permanent memory, in which are installed a main program and a virtual machine [which makes it possible]allowing to execute the main program and at least one program fragment using said microprocessor, [characterized in that]wherein said [on-board]embedded system includes at least one verification program module to [manage and]verify a downloaded program fragment in accordance with ~~the protocol for managing a downloaded program fragment as claimed in one of claims 1 to 3,~~a process including:

initializing the type stack and the table of register types through data representing the state of said virtual machine at the starting of the execution of said temporarily stored object code;

carrying out a verification process of said temporarily stored object code instruction by instruction, by discerning the existence, for each current instruction, of a target, a branching-instruction target, a target of an exception-handler call or a target of a subroutine call, and, said current instruction being the target of a branching instruction, said verification process consisting in verifying that the stack is empty and rejecting the program fragment otherwise;

carrying out a verification process and an updating of the effect of said current instruction on the data types of said type stack and of said table of register types;

said verification process being successful when the table of register types is not modified in the course of a verification of all the instructions, and said verification process being carried out

instruction by instruction until the table of register types is stable, with no modification being present, said verification process being interrupted and said program fragment being rejected, otherwise;

said management and verification program module being installed in the permanent memory.

Claim 21. (Canceled)

22. (Currently amended) A [method of]system for transforming an object code of a program fragment including a series of instructions, in which the operands of each instruction belong to the data types manipulated by [this]said instruction, the execution stack does not exhibit any overflow phenomenon[,] and for each branching instruction, the type of stack variables at [this]a corresponding branching is the same as [at]that of the targets of this branching, and an operand of given type written to a register by an instruction of [this]said object code is reread from [this]said same register by another instruction of this object code with the same given data type, into a standardized object code for this same program fragment, ~~in which the operands of each instruction belong to the data types manipulated by this instruction, the execution stack does not exhibit overflow phenomenon, the execution stack is empty at each branching instruction and at each branching target instruction, the same data type being assigned to the same register throughout said standardized object code, characterized in that~~wherein said [conversion]transforming system includes, at least, installed in the working memory of a development computer or workstation, a program module [to]for transforming [this]said object code into a standardized object code in accordance with ~~the method as claimed in one of claims 15 to 18, making it possible to generate a standardized object code for said~~

~~program fragment, satisfying the criteria for verifying this downloaded program fragment~~ a process of transforming including for all the instructions of said object code:

annotating each current instruction with the data type of the stack before and after execution of said current instruction, with the annotation data being calculated by means of an analysis of the data stream relating to said current instruction;

detecting, within said instructions and within each current instruction, the existence of branchings, or respectively of branching-targets, for which said execution stack is not empty, said detecting operation being carried out on the basis of the annotation data of the type of stack variables allocated to each current instruction; and, in case of detection of a non—empty execution stack,

inserting instructions to transfer stack variables on either side of said branchings or of said branching targets respectively, in order to empty the contents of the execution stack into temporary registers before said branching and to reestablish the execution stack from said temporary registers after said branching; and

not inserting any transfer instruction otherwise, said method allowing thus to obtain said standardized object code for said same program fragment, in which the operands of each instruction belong to the data types manipulated by said instruction, the execution stack does not exhibit any overflow phenomenon, the execution stack is empty at each branching instruction and at each branching-target instruction, in the absence of any modification to the execution of said program fragment.

Claim 23. (Canceled)

24. (Currently amended) A computer program product which is recorded on a medium and can be loaded directly from a terminal into the internal memory of a reprogrammable [on-board] embedded system[, such as a microprocessor card] equipped with a microprocessor and a rewritable memory, [this]said [on-board]embedded system making it possible to download and temporarily store a program fragment consisting of an object code[,] including a series of instructions, executable by [the] said microprocessor [of the on-board system]by way of a virtual machine equipped with an execution stack and with operand registers manipulated via [these]said instructions and making it possible to interpret [this]said object code, [this]said computer program product including portions of object code to execute the [stages]steps of verifying a program fragment downloaded onto [this]said [on-board]embedded system ~~as claimed in one of claims 4 to 14, when this on-board system is interconnected to a terminal and this program is executed by the microprocessor of this on-board system by way of said virtual machine~~according to a verifying process, said verifying process including:

initializing the type stack and the table of register types through data representing the state of said virtual machine at the starting of the execution of said temporarily stored object code;

carrying out a verification process of said temporarily stored object code instruction by instruction, by discerning the existence, for each current instruction, of a target, a branching-instruction target, a target of an exception-handler call or a target of a subroutine call, and, said current instruction being the target of a branching instruction, said verification process consisting in verifying that the stack is empty and rejecting the program fragment otherwise;

carrying out a verification process and an updating of the effect of said current instruction on the data types of said type stack and of said table of register types;

said verification process being successful when the table of register types is not modified in the course of a verification of all the instructions, and said verification process being carried out instruction by instruction until the table of register types is stable, with no modification being present, said verification process being interrupted and said program fragment being rejected, otherwise.

25. (Currently amended) A computer program product which is recorded on a medium including portions of object code to execute [stages]steps of [the method]a process of transforming an object code of a downloaded program fragment into a standardized object code for this same program fragment ~~as claimed in one of claims 15 to 18~~, said process of transforming including:

annotating each current instruction with the data type of the stack before and after execution of said current instruction, with the annotation data being calculated by means of an analysis of the data stream relating to said current instruction;

detecting, within said instructions and within each current instruction, the existence of branchings, or respectively of branching—targets, for which said execution stack is not empty, said detecting operation being carried out on the basis of the annotation data of the type of stack variables allocated to each current instruction; and, in case of detection of a non-empty execution stack, inserting instructions to transfer stack variables on either side of said branchings or of said branching targets respectively, in order to empty the

contents of the execution stack into temporary registers before said branching and to reestablish the execution stack from said temporary registers after said branching; and

not inserting any transfer instruction otherwise, said method allowing thus to obtain said standardized object code for said same program fragment, in which the operands of each instruction belong to the data types manipulated by said instruction, the execution stack does not exhibit any overflow phenomenon, the execution stack is empty at each branching instruction and at each branching—target instruction, in the absence of any modification to the execution of said program fragment.

26. (Currently amended) A computer program product which is recorded on a medium [which] and can be used in a reprogrammable [on-board]embedded system, [such as a microprocessor card]equipped with a microprocessor and a rewritable memory, [this]said [on-board]embedded system [making it possible]allowing to download a program fragment consisting of an object code, a series of instructions, executable by the microprocessor of [the]said [on-board]embedded system by [way]means of a virtual machine equipped with an execution stack and with local variables or registers manipulated via these instructions and making it possible to interpret [this]said object code, [this]said computer program product including, at least:

[-]program resources which can be read by the microprocessor of [this]said [on-board]embedded system via said virtual machine, to command execution of a procedure for managing the downloading of a downloaded program fragment;

[-]program resources which can be read by the microprocessor of [this]said [on-board]embedded system via said virtual machine, to command execution of a procedure for verifying, instruction by instruction, [the]said object code which makes up said program fragment;

[-]program resources which can be read by the microprocessor of [this]said [on-board]embedded system via said virtual machine, to command execution of a downloaded program fragment [following]subsequent to or in the absence of a conversion of [the]said object code of [this]said program fragment into a standardized object code for this same program fragment.

27. (Currently amended) The computer program product as claimed in claim 26, additionally including program resources which can be read by the microprocessor of [this]said [on-board]embedded system via said virtual machine, to command inhibition of execution, [on]by said [on-board]embedded system, of said program fragment in the case of an unsuccessful verification procedure of this program fragment.